

Wrangling osm data for Vilnius, Lithuania

Map area

Vilnius, Lithuania

<https://www.openstreetmap.org/relation/1529146#map=11/54.7010/25.2528&layers=HN>

The file:

vilniusmap.osm - 177,989 Kb

For this project I chose map data from Vilnius, the capital city of Lithuania. I am currently living here so this is the data set I wanted to explore

Important code files for this project:

- data.py - The main shaping file which takes my osm data, fixes it, and shapes it to CSV.
- importtosql.py - The file that contains the code to import contents from CSV file to database.
- TagTypes.py - Auditing file to check the types of tags, also checks for problems in those tags.
- audit.py - Auditing file to check all of the bad street types that I will have to fix.
- Auditpostcodes.py - Auditing post codes to check if it has LT- prefix.

Problems in the Map

After auditing my map file with TagTypes.py and audit.py, I found these problems:

- Checking my map file with TagTypes showed me that I had 222212 lowercase tags, 242214 tags that were separated by colon (addr:street), 685 other tags that consisted of numbers from 1 to 685 and 0 problematic characters. I decided that I will clean my data to split the tags with semicolon, so that the characters before the colon will be set as the tag type and the characters after the colon will be set as tag key.

```
type
{'lower': 222212, 'lower_colon': 242214, 'other': 685, 'problemchars': 0}

Process finished with exit code 0
```

- After printing some of the tags, I found that some of the street values consists of shortened street types. For example g. should mean gatvė in Lithuanian. I created a python script to audit it and check for the street types to let me know what street types I will need to change later in the data.py file. I will explain how I fixed it later in the document.

```

u'Senosios Pilait\u00117s kel.',
u'Suderv\u00117s kel.']],
u'pl.': set([u'Ei\u00161i\u00161ki\u00173 pl.',
'Minsko pl.',
u'Mol\u00117t\u00173 pl.',
u'Nemen\u0010din\u00117s pl.',
'Senasis Gardino pl.',
'Senasis Minsko pl.']],
'pr.': set([u'Balt\u00173 pr.',
'Gedimino pr.',
'Konstitucijos pr.',
u'Laisv\u00117s pr.',
u'Pilait\u00117s pr.',
u'Savanori\u00173 pr.']],
'skg.': set(['Baltasis skg.',

```

- I wanted to check if the post codes in my data set is in correct format. (The correct format for Lithuanian post codes are Symbols 'LT-' + five numbers according to this wikipedia page https://lt.wikipedia.org/wiki/ISO_3166-2). I wrote a script called Auditpostcodes.py to check for problematic post codes that doesn't include LT in front. I got the following results:

```
04213  
08444  
02120  
{ 'Bad': 1990, 'Good': 53, 'Irrelevant': 463068 }
```

- I could not import my data to sqlite3 database manually. All I got was this error INSERT failed: UNIQUE constraint failed. That could have been because of special lithuanian characters like ė, ą, ū etc. in my data. I found out that one of the solutions was to create a python script to import the data to the database.

Fixing colons in the tags

I had to split the `addr:street` key so that the "addr" would go to the type field and "street" to key field. To fix the colons I had to create a function that takes each element of the node and way tag, iterates it, find the key values matching the regex for a string with colon and splitting the string to 2 values: key and type.

```
def shape_element(element, node_attr_fields=NODE_FIELDS,
way_attr_fields=WAY_FIELDS,
problem_chars=PROBLEMCHARS, default_tag_type='regular'):

    node_attribs = {}
    way_attribs = {}
    way_nodes = []
    tags = [] # Handle secondary tags the same way for both node and way elements
    if element.tag == 'node':
        for attrib in element.attrib:
            if attrib in NODE_FIELDS:
                node_attribs[attrib] = element.attrib[attrib]
        for child in element:
            node_tag = {}
            if LOWER_COLON.match(child.attrib['k']): #Checks if the child attribute "key"
```

matches the regex.

```
node_tag["type"] = child.attrib["k"].split(':',1)[0]
```

After running this code my key attributes were clean and without colons.

Fixing Lithuanian street types

After auditing my data for the street types, I found out that mostly all of my street values consists of shortened street types. For example I had street types like g. for gatvė, al. for alėja, skg. for skersgatvis. To fix this I came up with an idea to include a new helper function to my data shaping file data.py. The helper function checked if the values of node_tag['value'], way_tag['value'] matches regex values for all of the street types, if it does, then it replaces the wrong street type with the right one. The helper function fix_streets is called in shape_element function.

```
for child in element:
    way_tag = {}
    way_node = {}
    if child.tag == 'tag':
        if LOWER_COLON.match(child.attrib['k']):
            way_tag["type"] = child.attrib["k"].split(':',1)[0]
            way_tag["key"] = child.attrib["k"].split(':',1)[1]
            way_tag["id"] = element.attrib["id"]
            way_tag["value"] = fix_streets(child.attrib['v'])
            print way_tag["value"] #Checking if it works
            tags.append(way_tag)
        elif PROBLEMCHARS.match(child.attrib['k']):
            continue
        else:
            way_tag["type"] = 'regular'
            way_tag["key"] = child.attrib["k"]
            way_tag["id"] = element.attrib["id"]
            way_tag["value"] = fix_streets(child.attrib['v'])
            tags.append(way_tag)
    elif child.tag == 'nd':
        way_node["id"] = element.attrib["id"]
        way_node["node_id"] = child.attrib["ref"]
        way_node["position"] = position
        position += 1
        way_nodes.append(way_node)
    return {'way': way_attrbs, 'way_nodes': way_nodes, 'way_tags': tags}
print tags
```

```
# ===== #
#           Helper Functions           #
# ===== #
#This function takes an attribute of a node, of way, then checks it against RegEx and if
```

it matches, it will replace the words.

```
def fix_streets(attribute):
    if FIND_GATVE.match(attribute):
        return attribute.replace('g.', 'gatvė'.decode("utf-8"))
    elif FIND_ALEJA.match(attribute):
        return attribute.replace('al.', 'alėja'.decode("utf-8"))
    elif FIND_AIKSTE.match(attribute):
        return attribute.replace('a.', 'aikštė'.decode("utf-8"))
    elif FIND_AKLIGATVIS.match(attribute):
        return attribute.replace('aklg.', 'akligatvis')
    elif FIND_KELIAS.match(attribute):
        return attribute.replace('kel.', 'kelias')
    elif FIND_PLENTAS.match(attribute):
        return attribute.replace('pl.', 'plentas')
    elif FIND_PROSPEKTAS.match(attribute):
        return attribute.replace('pr.', 'prospektas')
    elif FIND_SKERSGATVIS.match(attribute):
        return attribute.replace('skg.', 'skersgatvis')
    else:
        return attribute
```

Fixing this resulted in cleaner data for my street types.

Fixing post codes.

To audit the data set I wrote a script to look for matching values against 2 regexes: `LT[-\s]\d{5}` and `(^[0-9]{5}$)`. After getting the results I saw that I need to fix about 1990 values. To do so I changed my "fix_values" function in data.py file, so that it checks if `child.attrib['v']` matches my regex pattern and if it does, it adds 'LT-' in front of it.

```
POST_CODE = re.compile(r'^[0-9]{5}$')

elif POST_CODE.match(attribute):
    return attribute.replace(attribute, 'LT-' + attribute)
```

This fixed my postcodes to match Lithuanian standarts.

Importing to database.

For some time I was not able to import my csv files to sqlite. I tried importing the schema, created tables manually, but nothing worked. I found out that 2 of these solutions would work:

- Removing header lines from CSV files
- Write a script to import the CSV's After reading this discussion on udacity forums
<https://discussions.udacity.com/t/insert-failed-unique-constraint-failed-nodes-id/1>

[99847/2](#) I realised that It might be something to do with my non ASCII characters and that I should try to write a script to add the values.

- After the script was done, I was still getting 'ASCII' errors, and find out that decoding the data using `.decode("utf-8")` helped. This is a snippet of code from my `importtosql.py` file:

- ```
Read in the csv file as a dictionary, format the
data as a list of tuples:
with open('nodes.csv','rb') as fin:
 dr = csv.DictReader(fin) # comma is default delimiter
 to_db = [(i['id'], i['lat'],i['lon'], i['user'].decode("utf-8"), i['uid'],
i['changeset'], i['timestamp']) for i in dr]

with open('nodes_tags.csv','rb') as fin2:
 dr2 = csv.DictReader(fin2) # comma is default delimiter
 to_db2 = [(i['id'], i['key'].decode("utf-8"),i['value'].decode("utf-8"),
i['type']) for i in dr2]
 pprint(to_db2)

with open('ways.csv','rb') as fin3:
 dr3 = csv.DictReader(fin3) # comma is default delimiter
 to_db3 = [(i['id'], i['user'].decode("utf-8"),i['uid'].decode("utf-8"),
i['version'], i['changeset'], i['timestamp']) for i in dr3]

with open('ways_tags.csv','rb') as fin4:
 dr4 = csv.DictReader(fin4) # comma is default delimiter
 to_db4 = [(i['id'], i['key'].decode("utf-8"),i['value'].decode("utf-8"),
i['type']) for i in dr4]

with open('ways_nodes.csv','rb') as fin5:
 dr5 = csv.DictReader(fin5) # comma is default delimiter
 to_db5 = [(i['id'], i['node_id'],i['position']) for i in dr5]

insert the formatted data
cur.executemany("INSERT INTO nodes(id, lat, lon, user, uid,
changeset, timestamp) VALUES (?, ?, ?, ?, ?, ?, ?);", to_db)
cur.executemany("INSERT INTO nodes_tags(id, key, value, type)
VALUES (?, ?, ?, ?);", to_db2)
cur.executemany("INSERT INTO ways(id, user, uid, version,
changeset, timestamp) VALUES (?, ?, ?, ?, ?, ?);", to_db3)
cur.executemany("INSERT INTO ways_tags(id, key, value, type)
VALUES (?, ?, ?, ?);", to_db4)
cur.executemany("INSERT INTO ways_nodes(id, node_id, position)
VALUES (?, ?, ?);", to_db5)
commit the changes
conn.commit()
```

- After my data was cleaned and imported to SQL I could start querying the database.

## Exploring the data

This section provides basic statistics about the dataset. For a database I am using SQLite.

### File sizes

|                     |       |                                      |
|---------------------|-------|--------------------------------------|
| 03/02/2017 09:10 PM | <DIR> | .                                    |
| 03/02/2017 09:10 PM | <DIR> | ..                                   |
| 03/02/2017 09:10 PM | <DIR> | .ipynb_checkpoints                   |
| 03/02/2017 08:33 PM |       | 1,546b audit.py                      |
| 03/02/2017 09:10 PM |       | 11,425b DAND_P3_Data_wrangling.ipynb |
| 03/02/2017 08:38 PM |       | 8,841b data.py                       |
| 03/02/2017 06:40 PM |       | 866b getusers.py                     |
| 03/02/2017 06:40 PM |       | 3,835b importtosql.py                |
| 03/02/2017 06:40 PM |       | 992b iterativeparsing.py             |
| 03/02/2017 06:40 PM |       | 21b README.md                        |
| 03/02/2017 06:40 PM |       | 2,578b schema.py                     |
| 03/02/2017 08:35 PM |       | 1,174b TagTypes.py                   |
| 03/02/2017 06:40 PM |       | 232b testregex.py                    |
| 02/25/2017 04:36 PM |       | 182,260,000b vilniusmap.osm          |

### Number of unique users

Query to get number of nodes:

```
SELECT COUNT(DISTINCT(users.uid))
FROM (SELECT uid FROM nodes UNION ALL SELECT uid FROM ways) users;
```

Results in 656

### Number of Nodes

Query to get number of nodes:

```
SELECT COUNT(*) FROM nodes;
```

Results in 791572

### Number of Ways

```
SELECT COUNT(*) FROM ways;
```

Results in 124591

### List of top 10 types of shops in this region

```
SELECT value, COUNT(*) FROM nodes_tags WHERE key = "shop" GROUP BY value
ORDER BY COUNT(*) DESC LIMIT 10;
```

Results in

```
hairdresser|104
supermarket|82
kiosk|75
car_repair|73
clothes|69
convenience|63
alcohol|40
florist|35
bakery|34
books|30
```

### Count of contributions by specific user "jurkis"

```
SELECT e.user, COUNT(*) FROM (SELECT user FROM nodes UNION ALL SELECT
user FROM ways) e WHERE user="Jurkis";
```

Results in Jurkis|57741

### Top node tags added by this user

```
SELECT e.user,e.key, COUNT(*) FROM (nodes JOIN nodes_tags ON nodes.id =
nodes_tags.id) e WHERE user = 'Jurkis' GROUP BY key ORDER BY COUNT(*) DESC
LIMIT 20;
```

Results in

```
Jurkis|name|193
Jurkis|amenity|132
Jurkis|city|92
Jurkis|housenumber|91
Jurkis|street|91
Jurkis|website|80
Jurkis|opening_hours|76
Jurkis|shop|76
Jurkis|phone|68
Jurkis|highway|66
Jurkis|country|58
Jurkis|natural|46
Jurkis|crossing|45
Jurkis|operator|36
Jurkis|email|33
```

Jurkis|level|28  
Jurkis|cuisine|27  
Jurkis|wheelchair|23  
Jurkis|postcode|22  
Jurkis|type|16

## Additional ideas

### Fixing max speed values

After querying the database to find weird values, I found that a lot of maxspeed values are bad.

```
SELECT key, value, COUNT(*)
FROM ways_tags WHERE key="maxspeed" GROUP BY value ORDER BY COUNT(*)
DESC;
```

Returns:

| Key        | Value         | Count(*) |
|------------|---------------|----------|
| "maxspeed" | "50"          | "1865"   |
| "maxspeed" | "LT:urban"    | "1411"   |
| "maxspeed" | "sign"        | "780"    |
| "maxspeed" | "40"          | "356"    |
| "maxspeed" | "60"          | "335"    |
| "maxspeed" | "20"          | "285"    |
| "maxspeed" | "70"          | "194"    |
| "maxspeed" | "90"          | "134"    |
| "maxspeed" | "80"          | "101"    |
| "maxspeed" | "30"          | "62"     |
| "maxspeed" | "LT:rural"    | "37"     |
| "maxspeed" | "10"          | "35"     |
| "maxspeed" | "100"         | "33"     |
| "maxspeed" | "5"           | "9"      |
| "maxspeed" | "LT:motorway" | "9"      |
| "maxspeed" | "130"         | "6"      |
| "maxspeed" | "120"         | "2"      |
| "maxspeed" | "LT:zone40"   | "1"      |

In my opinion these values are important for some users, and programs since it might calculate the travel times according to max speeds and this information should be as clean as possible. Fixing this would be kind of difficult because I don't know where to get the missing values from. I could statistically calculate the averages of max speed for that type of area/road etc and fill it with "average value". This solution would be bad for programs or people that need accurate data like GPS etc.. but might be usefull just for general purposes.